# AD-A254 923

ITATION PAGE

| 1. AGENCY USE ONLY *(Leave Blank)* | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | | Final: 20 May 1992 |

**4. TITLE AND SUBTITLE**

Validation Summary Report: Irvine Compiler Corporation ICC Ada v7.4.0, VAXstation 3100 Model M38/VMS Version V5.3-1(Host) Intel i960MX in Hughes DMV running in tagged mode (bare machine with CHKSYS kernel version 104)(T)920520I1.11260

**5. FUNDING NUMBERS**

**6. AUTHOR(S)**

IABG-AVF
Ottobrunn, Federal Republic of Germany

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

IABG-AVF, Industrieanlagen-Betriebsgeselschaft
Dept. SZT/ Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

**8. PERFORMING ORGANIZATION REPORT NUMBER**

IABG-VSR 106

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**

DTIC
ELECTE
AUG 25 1992
S A D

**12a. DISTRIBUTION/AVAILABILITY STATEMENT**

Approved for public release; distribution unlimited.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

Irvine Compiler Corporation, ICC Ada v7.4.0, VAXstation 3100 Model M38/VMS Version V5.3-1 (Host) Intel i960MX in Hughes DMV running in tagged mode (bare machine with CHKSYS kernel version 104) (Target), 920520I1.11260

92 8 19 89

92-23164

391490   49 pgp

**14. SUBJECT TERMS**

Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

**15. NUMBER OF PAGES**

**16. PRICE CODE**

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION | 19 SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFED | UNCLASSIFIED | |

NSN 7540-01-280-550

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. 239-128

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 920520I1.11260
Irvine Compiler Corporation
ICC Ada v7.4.0
VAXstation 3100 Model M38/VMS Version V5.3-1 =>
Intel i960MX in Hughes DMV running in tagged mode
(bare machine with CHKSYS kernel version 104)

DTIC QUALITY INSPECTED 5

| Accesion For | |
|---|---|
| NTIS CRA&I | ☑ |
| DTIC TAB | ☐ |
| Unannounced | ☐ |
| Justification | |
| By | |
| Distribution / | |
| Availability Codes | |
| Dist | Avail and / or Special |
| A-1 | |

Prepared By:
IABG mbH, Abt. ITE
Einsteinstr. 20
W-8012 Ottobrunn
Germany

# Certificate Information

The following Ada implementation was tested and determined to pass ACVC
1.11. Testing was completed on May 20, 1992.

> **Compiler Name and Version:** ICC Ada v7.4.0
>
> **Host Computer System:** VAXstation 3100 Model M38/VMS Version V5.3-1
>
> **Target Computer System:** Intel i960MX in Hughes DMV running in tagged
> mode (bare machine with CHKSYS kernel version
> 104)

See Section 3.1 for any additional information about the testing
environment.

As a result of this validation effort, Validation Certificate
#920520I1.11260 is awarded to Irvine Compiler Corporation. This certificate
expires 24 months after ANSI approval of MIL-STD 1815B.

This report has been reviewed and is approved.

IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany


Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA   22311


Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC   20301

# DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.


## Declaration of Conformance


Customer:                        Irvine Compiler Corporation

Ada Validation Facility:         IABG mbH

ACVC Version:                    1.11


**Ada Implementation:**

Ada Compiler Name and Version:   ICC Ada v7.4.0

Host Computer System:            VAXstation 3100 Model M38
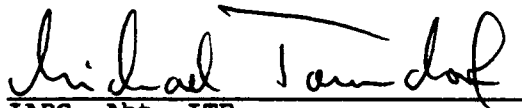                                 under VMS 5.3-1

Target Computer System:          Intel i960MX in Hughes DMV
                                 running in tagged mode (bare
                                 machine with CHKSYS kernel
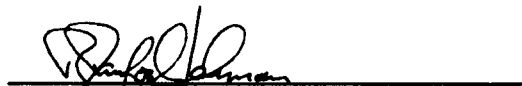                                 version 104)

**Declaration:**


I, the undersigned, declare that I have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.


_____          20 May 1992
Customer Signature                        Date

# TABLE OF CONTENTS

# CHAPTER 1

## INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

## 1.1  USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

> National Technical Information Service
> 5285 Port Royal Road
> Springfield VA  22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

> Ada Validation Organization
> Computer and Software Engineering Division
> Institute for Defense Analyses
> 1801 North Beauregard Street
> Alexandria VA  22311-1772

## 1.2 REFERENCES

[Ada83] *Reference Manual for the Ada Programming Language*,
ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] *Ada Compiler Validation Procedures*, Version 2.1, Ada Joint
Program Office, August 1990.

[UG89] *Ada Compiler Validation Capability User's Guide*, 21 June 1989.

## 1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC
contains a collection of test programs structured into six test classes:
A, B, C, D, E, and L. The first letter of a test name identifies the class
to which it belongs. Class A, C, D, and E tests are executable. Class B
and class L tests are expected to produce errors at compile time and link
time, respectively.

The executable tests are written in a self-checking manner and produce a
PASSED, FAILED, or NOT APPLICABLE message indicating the result when they
are executed. Three Ada library units, the packages REPORT and SPPRT13,
and the procedure CHECK_FILE are used for this purpose. The package REPORT
also provides a set of identity functions used to defeat some compiler
optimizations allowed by the Ada Standard that would circumvent a test
objective. The package SPPRT13 is used by many tests for Chapter 13 of the
Ada Standard. The procedure CHECK_FILE is used to check the contents of
text files written by some of the Class C tests for Chapter 14 of the Ada
Standard. The operation of REPORT and CHECK_FILE is checked by a set of
executable tests. If these units are not operating correctly, validation
testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class
B tests are not executable. Each test in this class is compiled and the
resulting compilation listing is examined to verify that all violations of
the Ada Standard are detected. Some of the class B tests contain legal Ada
code which must not be flagged illegal by the compiler. This behavior is
also verified.

Class L tests check that an Ada implementation correctly detects violation
of the Ada Standard involving multiple, separately compiled units. Errors
are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by
implementation-specific values -- for example, the largest integer. A list
of the values used for this implementation is provided in Appendix A. In
addition to these anticipated test modifications, additional changes may be
required to remove unforeseen conflicts between the tests and
implementation-dependent characteristics. The modifications required for
this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.


## 1.4 DEFINITION OF TERMS

Ada Compiler     The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.

Ada Compiler Validation Capability (ACVC)     The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.

Ada Implementation     An Ada compiler with its host computer system and its target computer system.

Ada Joint Program Office (AJPO)     The part of the certification body which provides policy and guidance for the Ada certification system.

Ada Validation Facility (AVF)     The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.

Ada Validation Organization (AVO)     The part of the certification body that provides technical guidance for operations of the Ada certification system.

Compliance of an Ada Implementation     The ability of the implementation to pass an ACVC version.

Computer System     A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity — Fulfillment by a product, process or service of all requirements specified.

Customer — An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.

Declaration of Conformance — A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.

Host Computer System — A computer system where Ada source programs are transformed into executable form.

Inapplicable test — A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.

ISO — International Organization for Standardization.

Operating System — Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.

Target Computer System — A computer system where the executable form of Ada programs are executed.

Validated Ada Compiler — The compiler of a validated Ada implementation.

Validated Ada Implementation — An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].

Validation — The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.

Withdrawn test — A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

# CHAPTER 2

## IMPLEMENTATION DEPENDENCIES

## 2.1  WITHDRAWN TESTS

The following tests have been withdrawn by the AVO.  The rationale for withdrawing each test is available from either the AVO or the AVF.  The publication date for this list of withdrawn tests is 02 August 1991.

| | | | | | |
|---|---|---|---|---|---|
| E28005C | B28006C | C32203A | C34006D | C35508I | C35508J |
| C35508M | C35508N | C35702A | C35702B | B41308B | C43004A |
| C45114A | C45346A | C45612A | C45612B | C45612C | C45651A |
| C46022A | B49008A | B49008B | A74006A | C74308A | B83022B |
| B83022H | B83025B | B83025D | B83026B | C83026A | C83041A |
| B85001L | C86001F | C94021A | C97116A | C98003B | BA2011A |
| CB7001A | CB7001B | CB7004A | CC1223A | BC1226A | CC1226B |
| BC3009B | BD1B02B | BD1B06A | AD1B08A | BD2A02A | CD2A21E |
| CD2A23E | CD2A32A | CD2A41A | CD2A41E | CD2A87A | CD2B15C |
| BD3006A | BD4008A | CD4022A | CD4022D | CD4024B | CD4024C |
| CD4024D | CD4031A | CD4051D | CD5111A | CD7004C | ED7005D |
| CD7005E | AD7006A | CD7006E | AD7201A | AD7201E | CD7204B |
| AD7206A | BD8002A | BD8004C | CD9005A | CD9005B | CDA201E |
| CE2107I | CE2117A | CE2117B | CE2119B | CE2205B | CE2405A |
| CE3111C | CE3116A | CE3118A | CE3411B | CE3412B | CE3607B |
| CE3607C | CE3607D | CE3812A | CE3814A | CE3902B | |

## 2.2  INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd.  For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

| | |
|---|---|
| C24113L..Y (14 tests) | C35705L..Y (14 tests) |
| C35706L..Y (14 tests) | C35707L..Y (14 tests) |
| C35708L..Y (14 tests) | C35802L..Z (15 tests) |
| C45241L..Y (14 tests) | C45321L..Y (14 tests) |
| C45421L..Y (14 tests) | C45521L..Z (15 tests) |
| C45524L..Z (15 tests) | C45621L..Z (15 tests) |
| C45641L..Y (14 tests) | C46012L..Z (15 tests) |

The following 20 tests check for the predefined type LONG_INTEGER; for this implementation, there is no such type:

| | | | | |
|---|---|---|---|---|
| C35404C | C45231C | C45304C | C45411C | C45412C |
| C45502C | C45503C | C45504C | C45504F | C45611C |
| C45613C | C45614C | C45631C | C45632C | B52004D |
| C55B07A | B55B09C | B86001W | C86006C | CD7101F |

C35713C, B86001U, and C86006G check for the predefined type LONG_FLOAT; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT; for this implementation, there is no such type.

A35801E checks that FLOAT'FIRST..FLOAT'LAST may be used as a range constraint in a floating-point type declaration; for this implementation, that range exceeds the range of safe numbers of the largest predefined floating-point type and must be rejected. (See section 2.3.)

C45423A..B (2 tests), C45523A, and C45622A check that the proper exception is raised if MACHINE_OVERFLOWS is TRUE and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOWS is FALSE.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45536A, C46013B, C46031B, C46033B, and C46034B contain length clauses that specify values for 'SMALL that are not powers of two or ten; this implementation does not support such values for 'SMALL.

B86001Y checks for a predefined fixed-point type other than DURATION; for this implementation, there is no such type.

CD1009C uses a representation clause specifying a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten TYPE'SMALL; this implementation does not support decimal 'SMALLs. (See section 2.3.)

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types; this implementation does not support such sizes.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions; this implementation provides no package MACHINE_CODE.

AE2101C and EE2201D..E (2 tests) use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types with discriminants without defaults; these instantiations are rejected by this compiler.

AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults; these instantiations are rejected by this compiler.

The tests listed in the following table check that USE_ERROR is raised if the given file operations are not supported for the given combination of mode and access method; this implementation supports these operations.

| Test | File Operation | Mode | File Access Method |
|------|----------------|------|--------------------|
| CE2102D | CREATE | IN_FILE | SEQUENTIAL_IO |
| CE2102E | CREATE | OUT_FILE | SEQUENTIAL_IO |
| CE2102F | CREATE | INOUT_FILE | DIRECT_IO |
| CE2102I | CREATE | IN_FILE | DIRECT_IO |
| CE2102J | CREATE | OUT_FILE | DIRECT_IO |
| CE2102N | OPEN | IN_FILE | SEQUENTIAL_IO |
| CE2102O | RESET | IN_FILE | SEQUENTIAL_IO |
| CE2102P | OPEN | OUT_FILE | SEQUENTIAL_IO |
| CE2102Q | RESET | OUT_FILE | SEQUENTIAL_IO |
| CE2102R | OPEN | INOUT_FILE | DIRECT_IO |
| CE2102S | RESET | INOUT_FILE | DIRECT_IO |
| CE2102T | OPEN | IN_FILE | DIRECT_IO |
| CE2102U | RESET | IN_FILE | DIRECT_IO |
| CE2102V | OPEN | OUT_FILE | DIRECT_IO |
| CE2102W | RESET | OUT_FILE | DIRECT_IO |
| CE3102E | CREATE | IN_FILE | TEXT_IO |
| CE3102F | RESET | Any Mode | TEXT_IO |
| CE3102G | DELETE | -------- | TEXT_IO |
| CE3102I | CREATE | OUT_FILE | TEXT_IO |
| CE3102J | OPEN | IN_FILE | TEXT_IO |
| CE3102K | OPEN | OUT_FILE | TEXT_IO |

CE2108B, CE2108D, CE2108F, CE2108H, CE3112B, CE3112D (6 tests) make checks that require the use of permanent files; this implementation does not support permanent files and so raises NAME_ERROR on the attempt to open a file created by an earlier-run test.

CE2203A checks that WRITE raises USE_ERROR if the capacity of an external sequential file is exceeded; this implementation cannot restrict file capacity.

CE2403A checks that WRITE raises USE_ERROR if the capacity of an external direct file is exceeded; this implementation cannot restrict file capacity.

CE3304A checks that SET_LINE_LENGTH and SET_PAGE_LENGTH raise USE_ERROR if they specify an inappropriate value for the external file; there are no inappropriate values for this implementation.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST; for this implementation, the value of COUNT'LAST is greater than 150000, making the checking of this objective impractical.

## 2.3  TEST MODIFICATIONS

Modifications (see section 1.3) were required for 50 tests and 2 support packages.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B24009A    B53001A    B59001E    B59001F    B83033B

C34009D and C34009J were graded passed by Evaluation Modification as directed by the AVO. These test check that 'SIZE for a composite type is greater than or equal to the sum of its components' 'SIZE values; but this issue is not addressed by AI-00825, which has not been considered; there is not an obvious interpretation. This implementation represents array components whose length depends on a discriminant with a default value by implicit pointers into the heap space; thus the 'SIZE of such a record type might be less than the sum of the components' 'SIZEs, since the heap space that is used by the varying-length array components is not counted as part of the 'SIZE of the record type. These tests were graded passed given that the Report.Result output was "FAILED" and the only Report.Failed output was "INCORRECT 'BASE'SIZE", from line 195 of C34009D and line 193 in C34009J.

A35801E was graded inapplicable by Evaluation Modification as directed by the AVO.  The compiler rejects the use of the range FLOAT'FIRST..FLOAT'LAST as the range constraint of a floating-point type declaration because the bounds lie outside of the range of safe numbers (cf. [Ada83] 3.5.7:12).

LA3004A and LA3004B were graded passed by Processing and Evaluation Modification as directed by the AVO. These tests check that when the bodies of library units (a procedure, function, and package) are made obsolete, the implementation will detect the missing bodies at link time. This implementation detects the missing bodies at link time, but it also issues error messages that indicate that the main procedure (resp.) must be recompiled. This behavior violates [Ada83] 10.3:6 & 8. To verify that the implementation does not in fact require recompilation of the main procedures, the obsolete bodies were recompiled (files LA3004A2..4 and LA3004B2..4 were modified to contain only the bodies) and the tests were then linked and executed. Report.Result output "NOT APPLICABLE" as expected.

The tests below were graded passed by Test Modification as directed by the AVO. These tests all use one of the generic support procedures Length_Check or Enum_Check (in the support files LENCHECK.ADA & ENUMCHEK.ADA), which use the generic procedure Unchecked_Conversion. This implementation rejects instantiations of Unchecked_Conversion with array types that have non-static index ranges. The AVO ruled that since this issue was not addressed by AI-00590, which addresses required support for Unchecked_Conversion, and since AI-00590 is considered not binding under ACVC 1.11, the support procedures were modified to remove the use of Unchecked_Conversion. Lines 40..43, 50, and 56..58 in LENCHECK and lines 42, 43, and 58..63 in ENUMCHEK were commented out.

| | | | | | |
|---|---|---|---|---|---|
| CD1009A | CD1009I | CD1009M | CD1009V | CD1009W | CD1C03A |
| CD1C04D | CD2A21A..C | CD2A22J | CD2A23A..B | CD2A24A | CD2A31A..C |
| CD2A81A | CD3014C | CD3014F | CD3015C | CD3015E..F | CD3015H |
| CD3015K | CD3022A | CD4061A | | | |

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of a power-of-10 value as 'SMALL for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal 'SMALLs may be omitted.

CE2102C, CE2102H, CE2103A..B, CE3102B, and CE3107A (6 tests) were graded passed by Evaluation Modification as directed by the AVO. Each of these tests contains a check that requires an illegal file name; but for this implementation, there are no illegal filenames. These tests are ruled to have been passed, since all of their applicable checks were indeed passed. The following Report.Failed output was generated:

```
CE2102C,
CE2102H:   NAME_ERROR NOT RAISED - CREATE [respectively: SEQ | DIR] 1
           NAME_ERROR NOT RAISED - CREATE [respectively: SEQ | DIR] 2
CE2103A,
CE2103B,
CE3107A:   NAME_ERROR NOT RAISED - UNSUCCESSFUL CREATE

CE3102B:   NO EXCEPTION RAISED FOR [some file name] - CREATE
           NO EXCEPTION RAISED FOR [some file name] - CREATE
           OTHER EXCEPTION RAISED FOR [some file name] - OPEN
           OTHER EXCEPTION RAISED FOR [some file name] - OPEN
```

CE2108B, CE2108D, CE2108F, CE2108H, CE3112B, and CE3112D were graded inapplicable by Evaluation Modification as directed by the AVO. These tests each depend on the contents of a file that was created by a previously executed test (e.g., CE2108B depends on a file created by CE2108A); but this implementation uses a file system that is implemented in the target computer's memory and does not support permanent files. The execution of tests CE2108F, CE2108H, and CE3112D results in a run-time system error report since NAME_ERROR is raised by the attempt to open the file from the previous test and is not handled. For tests CE2108B, CE2108D, and CE3112B, NAME_ERROR is similarly raised; it is handled anonymously and the following Report.Failed output is generated:

```
UNEXPECTED EXCEPTION RAISED ON OPENING OF TEXT FILE, WHICH SHOULD
HAVE BEEN CREATED BY TEST [respectively: CE2108A | CE2108C | CE3112A]
```

# CHAPTER 3

## PROCESSING INFORMATION

### 3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For technical and sales information about this Ada implementation, contact:

> Joe Kohli
> Irvine Compiler Corporation
> 34 Executive Park, Suite 270
> Irvine, California 92714 USA
> Tel. (714) 250-1366
> Fax. (714) 250-0676

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

### 3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system -- if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a) Total Number of Applicable Tests        3783
b) Total Number of Withdrawn Tests           95
c) Processed Inapplicable Tests              91
d) Non-Processed I/O Tests                    0
e) Non-Processed Floating-Point
            Precision Tests                  201

f) Total Number of Inapplicable Tests       292   (c+d+e)

g) Total Number of Tests for ACVC 1.11      4170  (a+b+f)


## 3.3  TEST EXECUTION

A magnetic tape containing the customized test suite (see section 1.3) was
taken on-site by the validation team for processing.  The contents of the
magnetic tape were loaded onto the host computer via ETHERNET (DEC DRQ3).

After the test files were loaded onto the host computer, the full set of
tests except 159 Floating-Point Precision Tests were processed by the Ada
implementation.

The tests were compiled and linked on the host computer system, as
appropriate.  The executable images were transferred to the target computer
system by the communications link described above, and run.  The results were
captured on the host computer system.

Testing was performed using command scripts provided by the customer and
reviewed by the validation team.  See Appendix B for a complete listing of
the processing options for this implementation.   It also indicates the
default options.  The options invoked explicitly for validation testing were

    -stack_check            enable stack overflow checking
    -numeric_check          enable arithmetic overflow checking
    -elaboration_check      enable elaboration checking
    -noinfo                 suppress informational messages
    -quiet                  suppress compiler banners
    -link=                  link the provided subprogram
    -listing                generate a compilation listing
    -maximum_errors=0       set maximum number of errors before abort to
                            unlimited
    -nopreprocess           disable compilation of preprocessor directives
    -nowarnings             suppress warnings
    -opt                    optimize code

Test output, compiler and linker listings, and job logs were captured on a
magnetic tape and archived at the AVF.  The listings examined on-site by the
validation team were also archived.

# APPENDIX A

## MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC.
The meaning and purpose of these parameters are explained in [UG89]. The
parameter values are presented in two tables. The first table lists the
values that are defined in terms of the maximum input-line length, which is
the value for $MAX_IN_LEN--also listed here. These values are expressed
here as Ada string aggregates, where "V" represents the maximum input-line
length.

| Macro Parameter | Macro Value |
|---|---|
| $MAX_IN_LEN | 254  -- Value of V |
| $BIG_ID1 | (1..V-1 => 'A', V => '1') |
| $BIG_ID2 | (1..V-1 => 'A', V => '2') |
| $BIG_ID3 | (1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A') |
| $BIG_ID4 | (1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A') |
| $BIG_INT_LIT | (1..V-3 => '0') & "298" |
| $BIG_REAL_LIT | (1..V-5 => '0') & "690.0" |
| $BIG_STRING1 | '"' & (1..V/2 => 'A') & '"' |
| $BIG_STRING2 | '"' & (1..V-1-V/2 => 'A') & '1' & '"' |
| $BLANKS | (1..V-20 => ' ') |
| $MAX_LEN_INT_BASED_LITERAL | "2:" & (1..V-5 => '0') & "11:" |
| $MAX_LEN_REAL_BASED_LITERAL | "16:" & (1..V-7 => '0') & "F.E:" |
| $MAX_STRING_LITERAL | '"' & (1..V-2 => 'A') & '"' |

The following table lists all of the other macro parameters and their respective values.

| Macro Parameter | Macro Value |
|---|---|
| $ACC_SIZE | 96 |
| $ALIGNMENT | 4 |
| $COUNT_LAST | 2147483647 |
| $DEFAULT_MEM_SIZE | 2097152 |
| $DEFAULT_STOR_UNIT | 8 |
| $DEFAULT_SYS_NAME | I960MX |
| $DELTA_DOC | 2#1.0#E-31 |
| $ENTRY_ADDRESS | address_of_entry1 |
| $ENTRY_ADDRESS1 | address_of_entry2 |
| $ENTRY_ADDRESS2 | address_of_entry3 |
| $FIELD_LAST | 2147483647 |
| $FILE_TERMINATOR | ' ' |
| $FIXED_NAME | NO_SUCH_FIXED_TYPE |
| $FLOAT_NAME | NO_SUCH_FLOAT_NAME |
| $FORM_STRING | "" |
| $FORM_STRING2 | "CANNOT_RESTRICT_FILE_CAPACITY" |
| $GREATER_THAN_DURATION | 524287.5 |
| $GREATER_THAN_DURATION_BASE_LAST | 10000000.0 |
| $GREATER_THAN_FLOAT_BASE_LAST | 1.0E+309 |
| $GREATER_THAN_FLOAT_SAFE_LARGE | 1.0E+308 |
| $GREATER_THAN_SHORT_FLOAT_SAFE_LARGE | 1.0E+38 |
| $HIGH_PRIORITY | 15 |
| $ILLEGAL_EXTERNAL_FILE_NAME1 | |

A-2

/NODIRECTORY/FILENAME

$ILLEGAL_EXTERNAL_FILE_NAME2
                  /NODIRECTORY/THIS-FILENAME-IS-ILLEGAL

$INAPPROPRIATE_LINE_LENGTH
                  -1

$INAPPROPRIATE_PAGE_LENGTH
                  -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE ("a28006d1.tst")

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("b28006d1.tst")

$INTEGER_FIRST          -2147483648

$INTEGER_LAST           2147483647

$INTEGER_LAST_PLUS_1    2147483648

$INTERFACE_LANGUAGE     C

$LESS_THAN_DURATION     -524287.5

$LESS_THAN_DURATION_BASE_FIRST
                  -10000000.00

$LINE_TERMINATOR        ASCII.LF

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                  NULL;

$MACHINE_CODE_TYPE
                  NO_SUCH_TYPE

$MANTISSA_DOC           31

$MAX_DIGITS             15

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2147483648

$MIN_INT                -2147483648

$NAME                   SHORT_SHORT_INTEGER

$NAME_LIST              I960,I960KA,I960KB,I960MC,I960CA,
                        I960SA,I960SB,I960MM,I960XA,I960MX

$NAME_SPECIFICATION1    X2120A

$NAME_SPECIFICATION2    X2120B

$NAME_SPECIFICATION3   X3119A

$NEG_BASED_INT         16#FFFFFFFE#

$NEW_MEM_SIZE          2097152

$NEW_STOR_UNIT         8

$NEW_SYS_NAME          I960MC

$PAGE_TERMINATOR       ASCII.FF

$RECORD_DEFINITION     NEW INTEGER;

$RECORD_NAME           NO_SUCH_MACHINE_CODE_TYPE

$TASK_SIZE             32

$TASK_STORAGE_SIZE     16383

$TICK                  (1.0/4096.0)

$VARIABLE_ADDRESS      address_of_var1

$VARIABLE_ADDRESS1     address_of_var2

$VARIABLE_ADDRESS2     address_of_var3

# APPENDIX B

## COMPILATION SYSTEM AND LINKER OPTIONS

The compiler and linker options of this Ada implementation, as described in
this Appendix, are provided by the customer. Unless specifically noted
otherwise, references in this appendix are to compiler documentation and not
to this report.

ICC Command Interpreter [v7.3.13 Apr 24, 1992] Help

Passive ICC Qualifiers
  /ARGUMENTS /ARGS              Display all arguments to the ICC command
  /DISPLAY                      Display all actions as they are performed
  /ENVIRONMENT=<arg>            Set the environment variable prefix
  /HELP                         List commonly used qualifiers
  /HELPALL /ALL                 List all available qualifiers
  /HIDE                         Suppress naming ICC subprocesses (VMS only)
  /IGNORE_CONFIGURATION /ICFG /IGNORE_CFG
                                Ignore configuration file qualifiers
  /IGNORE_ENVIRONMENT /IENV /IGNORE_ENV
                                Ignore environment variable qualifiers
  /NORMAL                       Compile with 'normal' messages
  /QUIET                        Compile quietly
  /SAVE_PAS2                    Save temporary files generated by pas2
  /SAVE_TEMPS /SAVE             Save all intermediate files
  /SUCCEED                      Always return the success status
  /SYMBOLS /SYMS                Show current value of ICC command's symbols
  /TEMP                         Use temporary directory for intermediate files
  /TMP=<arg>                    Use <arg> as the temporary directory
  /UNIQUE                       Use unique file names for intermediate files
  /VERBOSE /V                   Compile with verbose messages

Active ICC Qualifiers
  /ARCHITECTURE=<arg>           Specify i960 architecture KA,KB,MC,CA,SA,SB,MM
  /ASM                          Stop at the generated assembly file
  /ASM_FLAG /ASMF=<arg>         Explicitly add flag(s) for the assembler
  /ASM_NAME /ASMN=<arg>         Use <arg> as the assembler
  /C                            Stop at the generated C source file
  /CC_FLAG /CCF=<arg>           Explicitly add flag(s) for the C compiler
  /CC_NAME /CCN=<arg>           Use <arg> as the C compiler
  /CODE_GENERATOR_MODE /CGMODE=<arg>
                                <arg> specifies the code generator mode
  /COPTIMIZE                    Invoke the C optimizer
  /EXE                          Link a non-Ada program
  /EXECUTE                      Execute and delete executable after linking
  /EXECUTE_FLAG /EXECF=<arg>    Explicitly add flag(s) for the executable
  /ADA_EXT=<arg>                Set Ada file extension
  /ASM_EXT=<arg>                Set assembly file extension
  /C_EXT=<arg>                  Set C file extension
  /EXE_EXT=<arg>                Set executable file extension
  /INT_EXT=<arg>                Set IFORM file extension
  /LIB_EXT=<arg>                Set object archive/library file extension
  /MRG_EXT=<arg>                Set list merge file extension
  /OBJ_EXT=<arg>                Set object file extension
  /OPT_EXT=<arg>                Set optimized IFORM file extension
  /INT                          Stop at the generated iform file
  /KEEP_TEMPS=<arg>             Save file(s) with extension(s) in <arg>
  /LIBRARY /LIB=<arg>           Set the compilation library directory
  /LINKFILE_NAME=<arg>          Set the linker directive file to <arg>
  /LINKER_NAME /LINKER /LOADER_NAME /LOADER=<arg>
                                Use <arg> as the linker
  /LINKER_PREFLAG /LINKPREF /LOADER_PREFLAG /LOADPREF=<arg>
                                Explicitly add pre-flag(s) for linker
  /LINKER_POSTFLAG /LINKPSTF /LOADER_POSTFLAG /LOADPSTF=<arg>
                                Explicitly add post-flag(s) for linker
  /MAP=<arg>                    Generate a link map file (ICC linker only)
  /MERGE                        Invoke the ICC list merger
  /MRG                          Stop at the generated list merge file
  /OBJ                          Stop at the generated object file [default]
  /OBJLIB=<arg>                 Install the object file in library <arg>
  /OBJLIB_FLAG /OBJLIBF=<arg>   Explicitly add flag(s) for object librarian
  /OBJLIB_NAME /OBJLIBN=<arg>   Use <arg> as the object librarian
  /OPT                          Stop at the optimizer iform file
  /OPTIMIZE_OPTIONS /OPT_OPT /OPTO=<arg>

```
                                   Specify optimization options
/OPTIMIZE /OPT                     Invoke the Ada optimizer
/PAGE_LENGTH=<arg>                 Set globally applicable page length
/PRELINKER /PRELOADER=<arg>        Execute <arg> before linking
/PRELINKER_FLAG /PRELINKF /PRELOADER_FLAG /PRELOADF=<arg>
                                   Explicitly add flag(s) for prelinker
/POSTLINKER /POSTLOADER=<arg>
                                   Execute <arg> after linking
/POSTLINKER_FLAG /PSTLINKF /POSTLOADER_FLAG /PSTLOADF=<arg>
                                   Explicitly add flag(s) for postlinker
/RANLIB_NAME /RANLIBN=<arg>        Use <arg> as the ranlib library processor
/RELEASE=<arg>                     Set the release directory
/SHOW_ONLY                         Display all actions to be performed
/SYSTEM=<arg>                      Set the system library directory
/TOOL_VERSION=<arg>                Specify the ICC toolset version

Ada Qualifiers
/ADVISE                            Enable advisory warnings [default]
/CHECKS                            Enable all runtime checks [default]
/COMPATIBLE_CALLS                  Generate calls compatible with C calls
/CROSS_REFERENCE /XREF             Generate cross-reference file (.xrf)
/DECLARE=<arg>                     Declare an identifier
/DEBUGGER                          Compile for the Ada symbolic debugger
/ELABORATION_CHECK /ELAB_CHECK
                                   Generate ELABORATION checking
/EXCEPTION_INFO                    Enable extra EXCEPTION information [default]
/INFORMATION                       Enable informational warnings [default]
/LISTING                           Generate list file (.lst)
/LISTING_OPTIONS /LIST_OPT /LISTO=<arg>
                                   Specify list file options
/MAXIMUM_ERRORS=<arg>              Set maximum number of errors reported
/PREPROCESS                        Generate commented preprocess file (.app)
/RATE                              Rate code efficiency
/RUNTIME                           Generate Ada runtime calls [default]
/STACK_CHECK                       Generate stack checking code
/STATIC_ELABORATION                Perform static elaboration [default]
/SYNTAX_ONLY                       Syntax check only
/TRIM                              Generate trimmed preprocess file (.app)
/WARNINGS /W                       Enable warnings
/WRAP                              Enable auto-wrapping error messages [default]
/ZERO                              Zero all records [default]

ICC Code Generator Qualifiers
/BRANCH_RELATIVE                   Use relative branches (BSO only)
/CONST_IN_CODE                     Place constant aggregates in CODE segment
/HOSTDEBUGGER /DBX /CDB /XDB       Generate host debugger information
/EXTENDED_LISTING /EXL /ASL        Generate extended code listing output
/EXTENDED_LISTING_OPTIONS /EXL_OPT /EXLO=<arg>
                                   Specify extended code listing options
/FPA                               Generate inline hardware FPA code (HPUX only)
/LEAF_PROCEDURES                   Generate LEAF procs when possible (1960 only)
/GPROFILE                          Generate runtime 'gprof' profiling
/LOC_INFO                          Generate extended local information
/FPU /M68881 /M68882               Generate inline hardware floating-point code
/FRAME_SIZE                        Generate frame size for each subp (1960 only)
/NAMES                             Generate namelist file (.n)
/NUMERIC_CHECK /NUMCHK             Generate overflow detection code
/PROBE_STACK                       Generate stack probes
/PROFILE                           Generate runtime profiling
/REAL                              Use real names
/C_STATIC                          Static mode (C code generator only)

ICC Prelinker Qualifiers
/COMPLINK /CL=<arg>                Ada compile and link <arg> into one file
/FORCE_LINK                        Force link, even if dependency errors
/HEAP_SIZE=<arg>                   Allocate <arg> bytes of heap (1960 only)
```

```
/IMPLICIT_HANDLER          Link main with implicit exception table
/LINK /L=<arg>             Ada link compilation unit <arg>
/OUTPUT /O=<arg>           Use <arg> as the executable file name
/STACK_SIZE /STACK=<arg>   Allocate <arg> bytes of user stack (1960 only)
/TRAP                      Establish numeric fault trap handler
/VMSDEBUGGER               Link with VAX/VMS Debugger
```

# APPENDIX C

## APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to
implementation-dependent pragmas, to certain machine-dependent conventions
as mentioned in Chapter 13 of the Ada Standard, and to certain allowed
restrictions on representation clauses.  The implementation-dependent
characteristics of this Ada implementation, as described in this Appendix,
are provided by the customer.  Unless specifically noted otherwise,
references in this Appendix are to compiler documentation and not to this
report.  Implementation-specific portions of the package STANDARD, which
are not a part of Appendix F, are given on the following lines.

```
package STANDARD is

    type SHORT_SHORT_INTEGER  is range -128 .. 127;
    type SHORT_INTEGER        is range -32768 .. 32767;
    type INTEGER              is range -2147483648 .. 2147483647;

    type FLOAT is digits 15 range
        -1.79769313486232E+308 .. 1.79769313486232E+308

    type SHORT_FLOAT is digits 6 range
        -3.40282E+38 .. 3.40282E+38

    type DURATION is delta 2.0**(-12) range -524287 .. 524287;

    ...

end STANDARD;
```

# Appendix F
# ICC Ada Version 7.4
# VAX / VMS to Intel i960™ MX / Bare

Irvine Compiler Corporation
34 Executive Park, Suite 270
Irvine, CA 92714
(714) 250-1366

May 20, 1992

## 1   ICC Ada Implementation

The Ada language definition leaves implementation of certain features to
the language implementor. This appendix describes the implementation-
dependent characteristics of ICC Ada.

## 2   Pragmas

The following predefined pragmas are implemented in ICC Ada as described
by the Ada Reference Manual (LRM):

**Elaborate** This pragma allows the user to modify the elaboration order of
compilation units.

**Inline** Subprogram inlining is implemented. Inline substitutions are per-
formed by the ICC optimizer. This pragma is not supported for generic
subprograms or subprograms which contain nested subprograms.

**List** This pragma enables or disables writing to the output list file.

**Optimize** is a predefined pragma which has been implemented for both **SPACE** and **TIME**. Specifying this pragma currently does *not* control whether the ICC optimizer is called; however, it does impact which optimizations are performed.

**Pack** Packing on arrays and records is implemented to the bit level. Slices of packed arrays are not implemented, except boolean arrays.

**Page** This pragma ejects a new page in the output list file (if enabled).

**Priority** This pragma sets the priority of a task or main program. The range of the subtype **priority** is 0..15.

The following predefined pragmas have been extended by ICC:

**Interface** This pragma is allowed to designate variables in addition to subprograms. It is also allowed to have an optional third parameter which is a string designating the name for the linker to use to reference the variable or subprogram. The third parameter has the same effect as pragma **Interface_name**.

**Suppress** In addition to suppressing the standard checks, ICC also permits suppressing the following:

**Assertion_check** Suppressing **Assertion_check** suppresses the generation of assertion checking code that is emitted when using the ICC-defined package **ASSERTION**.

**Exception_info** Suppressing **Exception_info** improves run-time performance by reducing the amount of information maintained for messages that appear when exceptions are propagated out of the main program or any task.

**All_checks** Suppressing **All_checks** suppresses all of the standard checks as well as **Exception_info** and **Assertion_check**.

The following predefined pragmas are currently not implemented by ICC:

**Controlled**

**Memory_size**

**Shared**

> **Storage_unit**
>
> **System_name**

The following additional pragmas have been defined by ICC: (For further details on these pragmas refer to the *ICC Ada User's Reference Guide.*)

**Access_address** This pragma allows the user to indicate how an access type will be physically represented. The first parameter specifies the name of an access type. The second parameter specifies the type of address. Valid types are **linear**, **virtual**, and **ad**. Linear addresses are 32-bits wide. Virtual addresses are 64-bits wide and consist of an access descriptor (AD) and an offset. AD addresses are 32-bits wide and consist of an access descriptor with an implicit offset of zero.

**Compatible_calls** This pragma is used to specify that pass-by-reference parameter passing should be used for OUT and IN OUT scalar parameters. By default some of the ICC code generators use copy-in/copy-back for scalar OUT and IN OUT parameters. This pragma allows pass-by-reference calls to be performed (primarily for downward compatibility).

**Compress** This pragma reduces the storage required for discrete subtypes in structures (arrays and records). Its single argument is the name of a discrete subtype. It specifies that the subtype should be represented as compactly as possible (regardless of the representation of the subtype's base type) when the subtype is used in a structured type. The storage requirement for variables and parameters is not affected. Pragma **Compress** must appear prior to any reference to the named subtype.

**Export** This pragma is a complement to the predefined pragma **Interface**. It enables subprograms written in Ada to be called from other languages. It takes two or three arguments. The first is the language to be called from, the second is the subprogram name, and the third is an optional string designating the actual subprogram name to be used by the linker. Pragma **Export** must appear prior to the body of the designated subprogram.

**External_name** This pragma is equivalent to the ICC pragma **Export** with an implicit language type of "Ada" and a required external name.

This pragma allows the user to specify the exact name of the subprogram that will be used in the generated object file. This pragma is provided for compatibility with existing Ada source files.

**Foreign** This pragma is used to add an object file or an object library file to the link command line used when linking the current compilation unit. Pragma **Foreign** is most frequently used in conjunction with pragma **Interface** so that foreign object files may be automatically included when the Ada compilation unit is linked. This pragma accepts two parameters. The first parameter indicates the *location* of the foreign object name on the link command line. It must be either **Normal** or **Post**. The second parameter is a string denoting the foreign object. This string is passed unmodified to the linker, so it should be a complete filename. If the location is **Normal**, then the foreign object is included immediately after the current Ada compilation unit on the link command line. If the location is **Post**, then the foreign object name is included at the end of the link command line. When multiple **Foreign Post** pragmas are used in a single program, the order of the foreign objects on the link command line is not defined.

**Interface_constant** allows access to externally defined constant values. It can only be applied to scalar variables and is similar to pragma **Interface** except that it interfaces to a *constant* or *address* rather than to a *value*.

**Interface_Name** This pragma takes a variable or subprogram name and a string to be used by the linker to reference the variable or subprogram. It has the same effect as the optional third parameter to pragma **Interface**.

**Interrupt_handler** This pragma is used when writing procedures that will be invoked as interrupt handlers (independent of the tasking runtime). It does not have any parameters and must appear immediately within the declarative part of a procedure. The presence of this pragma causes the code generator to produce additional code on procedure entrance and exit which preserves the values of all global registers. This pragma has no other effect.

**Noinline** allows the programmer to specify that a subprogram *never* be inlined. This suppresses auto-inlining for the named subprogram. The parameter to pragma **Noinline** is a list of subprogram names. This

pragma is useful for machine code insertions where a call must always be performed.

**No_zero** The single parameter to **No_zero** is the name of a record type. If the named record type has *holes* (or *gaps*) between fields that are normally initialized with zeroes, this pragma will suppress the clearing of the holes. If the named record type has no holes this pragma has no effect. When zeroing is disabled, comparisons (equality and non-equality) of the named type are disallowed. The use of this pragma can significantly reduce initialization time for record objects. The ICC Command Interpreter also has the qualifier /NO_ZERO which has the effect of implicitly applying pragma **No_zero** to all record types declared in the file.

**Optimize_options** allows the user to modify optimization defaults. This allows better control of the options used in various portions of a compilation unit. This pragma has two parameters, a comma-separated list of subprograms (optional), and a list of optimization options. The options are specified in named association format (i.e., option_name => value). The option(s) are applied only within the scope of the enclosing declarative region and any nested regions. If a list of subprograms is not supplied, the option(s) apply to all subprograms in the current compilation unit. Note that, like pragma **Suppress**, options specified by this pragma in a package specification are inherited by the package body.

The following table summarizes the available options and their values:

| Name | Value | Description |
|------|-------|-------------|
| Space | Boolean | Optimize for space |
| Time | Boolean | Optimize for time |
| Autoinline | Boolean | Enable/disable autoinlining |
| Peephole | All_code | Enable peephole optimizations for all code generated |
| | Machine_code | Enable peephole optimizations only for machine code insertions |
| Nopeephole | All_code | Disable peephole optimizations |
| | Machine_code | Disable peephole optimizations only for machine code insertions |
| Inline | Boolean | Enable/disable explicit inlining |

**Protect_registers** This pragma is used with the I960_MACHINE_CODE package to specify a list of machine registers which should be saved on entrance to the current subprogram and restored on exit. This pragma is used primarily for low-level interrupt handlers.

**Put, Put_line** These pragmas take any number of arguments and write their value to standard output at compile time when encountered by the compiler. The arguments may be expressions of any string, enumeration, or integer type whose value is known at compile time. Pragma **Put_line** adds a carriage return after printing all of its arguments. These pragmas are often useful in conjunction with conditional compilation. They may appear anywhere a pragma is allowed.

**Reserve_registers** This pragma is used with the I960_MACHINE_CODE package to reserve a list of registers for machine code insertions. The listed registers will not be used by the ICC Optimizer or ICC Code Generator so that the programmer can use them in machine code insertions. Note: Do not attempt to reserve the i960 special registers (i.e., r0, r1, r2, and g15). This may cause error messages in the code generator. Reserving a large number of registers may cause the code generator to run out scratch registers and will cause the compilation to fail.

**Simple_task** is used to eliminate the normal dynamic tasking runtime associated with a task and request that the task be elaborated statically by the ICC i960 Initial Memory Image Builder. This pragma must appear within an entryless specification of a top-level task. Use of pragma **Simple_task** requires that there not be any operations performed that would require importing of the tasking runtime.

Simple tasks (i.e., those to which pragma **Simple_task** has been applied) may not declare entries and may not perform entry calls. The pragmas **Time_slice** and **Time_slice_attributes** may be applied to simple tasks to modify their runtime behavior.

**Static_elaboration** This pragma is used immediately within a package specification to state that all elaboration for the package is intended to be static. NOTE: This pragma does *not* modify the behavior operation of the Ada compiler; in particular, it does *not* enable static exceptions. This pragma only conveys information to the compiler that the user's intention is that all elaboration for the package be static. Warnings

will be generated for all objects within the package specification or corresponding body which require dynamic elaboration; however, the compiler will still compile those packages statically.

**Time_slice** may be used only in a task which has pragma **Simple_Task** applied to it. Pragma **Time_Slice** accepts one parameter: the period of a time slice. This parameter is of the predefined type Duration.

**Time_slice_attributes** may be used only in a task which has pragma **Simple_Task** applied to it. This pragma accepts one parameter: a list of initial values to be applied to the task's Process Controls.

**Unsigned_Literal** This pragma, when applied to a 32-bit signed integer type, affects the interpretation of literals for the type. Literals between $2^{**}31$ and $(2^{**}32)-1$ are accepted for the type and are represented as if the type were unsigned. Operations on the type are unaffected. Note that (with checking suppressed) signed addition, subtraction, and multiplication are equivalent to the corresponding unsigned operations. However, division and relational operators are different and should be used with caution.

**Uselib** This pragma is used within a context clause to explicitly add a list of named search libraries to the library search list of the current compilation. The specified libraries are searched *first* in all following WITH clauses.

**Volatile** notifies the ICC Optimizer and Code Generator peepholer that a scalar variable's value may change outside the scope of the program. Pragma **Volatile** accepts one parameter: the global scalar variable to be considered *volatile*. The use of this pragma suppresses value-following so that every reference to the specified variable always generates a memory load or store instruction. This is useful for memory mapped I/O devices where a global variable may be mapped directly to a device address.

# 3 Preprocessor Directives

ICC Ada incorporates an integrated preprocessor whose directives begin with the keyword **Pragma**. They are as follows:

**Abort** This pragma causes the current compilation to be immediately halted. It is useful when unexpected circumstances arise inside conditionally compiled code.

**If, Elsif, Else, End** These preprocessor directives provide a conditional compilation mechanism. The directives **If** and **Elsif** take a boolean static expression as their single argument. If the expression evaluates to **False** then all text up to the next **End, Elsif,** or **Else** directive is ignored. Otherwise, the text is compiled normally. The usage of these directives is identical to that of the similar Ada constructs. These directives may appear anywhere pragmas are allowed and can be nested to any depth.

**Include** This preprocessor directive provides a compile-time source file inclusion mechanism. It is integrated with the library management system and the automatic recompilation facilities.

**Include_file** is used to distinguish include files from normal Ada compilation units. If the Ada frontend encounters this pragma in a source file which has not been included (via **Pragma Include**), compilation is aborted immediately. NOTE: This pragma does *not* modify the behavior of the Ada compiler. It only conveys information to the compiler that the source file is intended as an include file. Specification of this pragma in include files is optional; however, ICC recommends this pragma be included in all include files to assist the ICM make facility.

The results of the preprocessor pass, with the preprocessor directives deleted and the appropriate source code included, may be output to a file at compile-time. The preprocessor *may* be disabled by using the /NOPREPROCESS command-line qualifier, in which case the above directives are ignored.

# 4 Attributes

ICC Ada implements all of the predefined attributes, including the Representation Attributes described in section 13.7 of the LRM.

Limitations of the predefined attributes are:

**Address** This attribute cannot be used with a statement label or a task entry.

The implementation-defined attributes for ICC Ada are:

**Alignment** This attribute returns the bit alignment for the prefix type. The value returned is a universal integer.

**Architecture** This attribute is for internal use within the ICC runtime and returns a value depending on the i960 architecture being targeted.

**Declared** This attribute is used to test the state of a value that may have been provided on the command-line when the compiler was invoked. **Declared** is used primarily in conditional compilation code (in conjunction with the preprocessor directive **If**) to modify how the source is compiled based on a compile-time option. The prefix to **Declared** must be a discrete type. Its result is a value of this discrete type.

**Linear_address** This attribute is currently identical to the predefined attribute **Address**. In the future **Linear_address** will return the 32-bit linear address for i960 targets and **Address** will return a full 64-bit virtual address.

**Version, System, Target, CG_mode** These attributes are used by ICC for conditional compilation. The prefix must be a discrete type. The values returned vary depending on the target architecture and operating system.

# 5 Input/Output Facilities

Standard Input and Output for embedded targets is implemented through the serial communications link between the host and embedded system. The normal TEXT_IO calls PUT and GET can be used for console I/O.

## 5.1 Virtual File System

Since embedded systems typically do not have access to disk storage, file I/O operations are *simulated* through a RAM-based *virtual file system* implemented by a low-level runtime support package. This virtual file system

10

behaves like a disk-based file system, except that *files* do not persist between program executions. Therefore, it is not possible to write a virtual file with one program and then to read it using another one. Also, the virtual file system never raises the exception NAME_ERROR since all file names are considered legal.

The implementation-dependent specifications from TEXT_IO and DIRECT_IO are:

```
type COUNT is range 0 ..  INTEGER'LAST;

subtype FIELD is INTEGER range 0 ..  INTEGER'LAST;
```

## 5.2    FORM Parameter

ICC Ada implements the FORM parameter to the procedures OPEN and CREATE in DIRECT_IO, SEQUENTIAL_IO, and TEXT_IO to perform a variety of ancillary functions. The FORM parameter is a string literal containing parameters in the style of named parameter notation. In general the FORM parameter has the following format:

$field_1$ => $value_1$ [, $field_n$ => $value_n$ ]

where $field_i$ => $value_i$ can be

```
OPTION             =>   NORMAL
OPTION             =>   APPEND
PAGE_MARKERS       =>   TRUE
PAGE_MARKERS       =>   FALSE
READ_INCOMPLETE    =>   TRUE
READ_INCOMPLETE    =>   FALSE
MASK               =>   <9 character protection mask>
```

Each *field* is separated from its *value* by "=>" and each *field/value* pair is separated by a comma. Spaces may be added anywhere between tokens and case is insignificant. For example:

```
create( f, out_file, "list.data",
     "option => append, PAGE_MARKERS => FALSE, Mask => rwxrwx---");
```

The interpretation of the fields and their values is presented below.

**OPTION** Files may be opened for appending. This causes data to be appended directly onto the end of an existing file. The default is **NORMAL** which overwrites existing data. This field applies to **OPEN** in all three standard I/O packages. It has no effect if applied to procedure **CREATE**.

**PAGE_MARKERS** If **FALSE** then all **TEXT_IO** routines dealing with page terminators are disabled. They can be called, however they will not have any effect. In addition the page terminator character (^L) is allowed to be read with **GET** and **GET_LINE**. The default is **TRUE** which leaves page terminators active. Disabling page terminators is particularly useful when using **TEXT_IO** with an interactive device. For output files, disabling page terminators will suppress the page terminator character that is normally written at the end of the file.

**READ_INCOMPLETE** This field applies to **DIRECT_IO** and **SEQUENTIAL_IO** and dictates what will be done with reads of incomplete records. Normally, if a **READ** is attempted and there is not enough data in the file for a complete record, then **END_ERROR** or **DATA_ERROR** will be raised. By setting **READ_INCOMPLETE** to **TRUE**, an incomplete record will be read successfully and the remaining bytes in the record will be zeroed. Attempting a read after the last incomplete record will raise **END_ERROR**. The **SIZE** function will reflect the fact that there is one more record when the last record is incomplete and **READ_INCOMPLETE** is **TRUE**.

**MASK** Set a protection mask to control access to a file. The mask is the standard nine character string notation used by Unix. The letters cannot be rearranged or deleted so that the string is always exactly nine characters long. This applies to **CREATE** in all three standard I/O packages. The default is determined at runtime by the user's environment settings.

The letters in the **Mask** are used to define the **Read**, **Write**, and **eXecute** permissions for the User, Group, and World respectively. Wherever the appropriate letter exists, the corresponding privilege is granted. If a "-" (dash) is used instead, then that privilege is denied. For example if **Mask** were set to "rw-rw----" then read and write privileges are granted to the file owner and his/her group, but no world rights are given.

If a syntax error is encountered within the FORM parameter then the exception USE_ERROR is raised at the OPEN or CREATE call. Also, the standard function TEXT_IO.FORM returns the current setting of the form fields, including default values, as a single string.

# 6 Package SYSTEM

```
package SYSTEM is

    type SHORT_SHORT_ORDINAL is range 0..(2**8)-1;
    for  SHORT_SHORT_ORDINAL'size use 8;

    type SHORT_ORDINAL is range 0..(2**16)-1;
    for  SHORT_ORDINAL'size use 16;

    type ORDINAL is range 0..(2**32)-1;
    for  ORDINAL'size use 32;

    type NAME is (I960,
                  I960KA, I960KB, I960MC,
                  I960CA, I960SA, I960SB,
                  I960MM, I960XA, I960MX);

    SYSTEM_NAME  : constant NAME := i960MX;
    STORAGE_UNIT : constant := 8;              -- Storage unit size in bits.
    MEMORY_SIZE  : constant := 2*(2**20);      -- Bytes.
    MIN_INT      : constant := -2**31;
    MAX_INT      : constant := 2**31-1;
    MAX_DIGITS   : constant := 15;
    MAX_MANTISSA : constant := 31;
    FINE_DELTA   : constant := 2.0**(-31);
    TICK         : constant := 1.0/4096.0;     -- This is max for type
                                               -- DURATION.

    type LINEAR_ADDRESS is range 0..(2**32)-1;
    for  LINEAR_ADDRESS'SIZE use 32;

    type MIXED_WORD is range 0..(2**32)-1;
    for  MIXED_WORD'SIZE use 32;

    subtype PRIORITY is INTEGER range 0 .. 15;  -- 0 is default priority.
```

```
ELABORATION_PRIORITY: constant PRIORITY := PRIORITY'LAST;

-- Constants for the STWHEAP package

BITS_PER_BMU  : constant := 8;           -- Bits per basic machine unit.
MAX_ALIGNMENT : constant := 4;           -- Maximum alignment required.
MIN_MEM_BLOCK : constant := 1024;        -- Minimum chunk request size.

-- Constants for the HOST package

-- HOST_CLOCK_RESOLUTION changed to 128 microseconds to support any
-- clock rate (16,25,33Mhz) for the i960.  See GET_TIME in HOST
-- for how actual time is loaded.

HOST_CLOCK_RESOLUTION : constant := 128;     -- 128 microseconds
BASE_DATE_CORRECTION  : constant := 25_202;  -- Unix base date is 1/1/1970.

NULL_LINEAR_ADDRESS : constant LINEAR_ADDRESS := 0;

subtype ADDRESS is LINEAR_ADDRESS;

pragma PUT_LINE ("System name is ", SYSTEM_NAME);

NULL_ADDRESS : constant ADDRESS := 0;   -- Value of type ADDRESS
                                        -- equal to NULL.

end SYSTEM;
```

# 7  Limits

Most data structures held within the ICC Ada compiler are dynamically
allocated and, hence, have no inherent limit (other than available memory).
Some limitations are:

The maximum input line length is 254 characters.

The maximum number of tasks abortable by a single abort state-
ment is 64.

Include files can be nested to a depth of three.

The number of packages, subprograms, tasks, variables, aggregates, types, or labels which can appear in a compilation unit is unlimited.

The number of compilation units which can appear in one file is unlimited.

The number of statements per subprogram or block is unlimited.

Packages, tasks, subprograms, and blocks can be nested to any depth.

There is no maximum number of compilation units per library, nor any maximum number of libraries per library system.

There is no limit on the range of numeric literal values.

There is no limit on the accuracy of floating point literal expressions.

There are no limits on the depth, levels of parentheses, or the number of terms in an expression.

All compile-time computations are done using exact integer or rational arithmetic.

# 8  Numeric Types

## 8.1  Integer Types

ICC Ada supports three predefined integer types:

| Type | Value Range | Width |
|------|-------------|-------|
| SHORT_SHORT_INTEGER | −128..127 | 8 bits |
| SHORT_INTEGER | −32768..32767 | 16 bits |
| INTEGER | −2147483648..2147483647 | 32 bits |

In addition, unsigned 8-bit, 16-bit, and 32-bit integer types can be defined by the user via the SIZE length clause. Storage requirements for types can be reduced by using pragma **Pack** and record representation clauses; for subtypes, by using the ICC pragma **Compress**.

## 8.2 Ordinal Types

ICC Ada supports the declaration of ordinal types using the size length clause. The following ordinal types are defined in package SYSTEM:

| Type | Value Range | Width |
|------|-------------|-------|
| SYSTEM.SHORT_SHORT_ORDINAL | 0..255 | 8 bits |
| SYSTEM.SHORT_ORDINAL | 0..65535 | 16 bits |
| SYSTEM.ORDINAL | 0..4294967295 | 32 bits |

All predefined operators for signed integers are also defined for the ordinal types.

## 8.3 Floating and Fixed Point Types

Type Float and type Short_float are implemented and have the following attributes:

| Attribute | Value | |
|-----------|-------|---|
| | type FLOAT | type SHORT_FLOAT |
| Machine_rounds | FALSE | FALSE |
| Machine_overflows | FALSE | FALSE |
| Machine_radix | 2 | 2 |
| Machine_mantissa | 52 | 23 |
| Machine_emax | 1020 | 124 |
| Machine_emin | $-1021$ | $-125$ |
| Mantissa | 51 | 21 |
| Digits | 15 | 6 |
| Size | 64 | 32 |
| First | $-1.79769313486232E+308$ | $-3.40282E+38$ |
| Last | $1.79769313486232E+308$ | $3.40282E+38$ |
| Base'First | $-1.79769313486232E+308$ | $-3.40282E+38$ |
| Base'Last | $1.79769313486232E+308$ | $3.40282E+38$ |
| Emax | 204 | 84 |
| Safe_Emax | 1020 | 124 |
| Epsilon | $8.88178419700125E-16$ | $9.53674316406250E-07$ |
| Small | $1.94469227433161E-62$ | $2.58493941422821E-26$ |
| Safe_small | $4.45014771701440E-308$ | $2.35098870164458E-38$ |
| Large | $2.57110087081438E+61$ | $1.93428038904620E+25$ |
| Safe_large | $1.12355820928895E+307$ | $2.12676377913539E+37$ |

Fixed point types automatically assume the smallest storage size necessary to represent all of the model numbers with the indicated delta and range. The size of a fixed point type may be changed via the SMALL representation clause and the SIZE length clause. Unsigned fixed point types may be defined using the SIZE length clause.

NOTE: ICC Ada rounds fixed point values away from zero at the midpoint between integral values. On the i960, ICC Ada uses round-to-nearest-even for floating point types. This means that the same value will round to an integer *differently* depending on whether it is a fixed or floating point value. The following table details the rounding behavior of fixed and floating point types on the i960.

| Value | Fixed Round | Float Round |
|-------|-------------|-------------|
| 1.00  | 1           | 1           |
| 1.25  | 1           | 1           |
| 1.50  | 2           | 2           |
| 2.50  | 3           | 2           |
| 2.75  | 3           | 3           |
| 3.50  | 4           | 4           |
| −1.25 | −1          | −1          |
| −1.50 | −2          | −2          |
| −1.75 | −2          | −2          |
| −2.50 | −3          | −2          |

## 8.4  Other Numeric Types

The ICC-defined type SYSTEM.ADDRESS is represented as a 32-bit unsigned integer. This allows address calculations to be performed as unsigned values and also allows the use of unsigned 32-bit literals.

The ICC-defined type SYSTEM.MIXED_WORD is a special 32-bit ordinal type which preserves the tag (33rd) bit on the i960 processor.

# 9  Tasks

The type DURATION is defined with the following characteristics:

| Attribute | Value |
|---|---|
| Delta | $2.44140625E - 04$ sec |
| Small | 0.0002 sec |
| First | $-524287.0$ sec |
| Last | 524287.0 sec |

The subtype SYSTEM.PRIORITY as defined provides the following range:

| Attribute | Value |
|---|---|
| First | 0 |
| Last | 15 |

Higher numbers correspond to higher priorities. If no priority is specified for a task, PRIORITY'FIRST is assigned during task creation.

# 10    Representation Clauses

## 10.1    Type Representation Clauses

### 10.1.1    Length Clauses

The amount of storage to be associated with an entity is specified by means of a length clause. The following is a list of length clauses and their implementation status:

- The SIZE length clause is implemented. When applied to integer-range types this length clause can be used to reduce storage requirements, including storage of unsigned values. It *may* be used to declare an unsigned 32-bit type. Length clauses are allowed for float and fixed point types, however the storage requirements for these types cannot be reduced below the smallest applicable predefined type available.

- The STORAGE_SIZE length clause for task types is implemented. The size specified is used to allocate both the task's Task Information Block (TIB) and its stack.

- The STORAGE_SIZE length clause for access types is implemented. When a length clause is encountered for an access type, a block of

memory is reserved in the user's heap space. This block of memory cannot be expanded beyond the bounds specified in the length clause. When the memory in this block is exhausted, STORAGE_ERROR is raised. Due to heap management overhead, the full amount of memory indicated in the length clause may not be available for allocation.

- The SMALL length clause for fixed point types is implemented for powers of two. ICC Ada does not support SMALL values that are not integral powers of two.

### 10.1.2  Enumeration Representation Clauses

Enumeration representation clauses are implemented. The use of enumeration representation clauses can greatly increase the overhead associated with their reference. In particular, FOR loops on such enumerations are very expensive. Representation clauses which define the default representation (i.e., the first element is ordinal 0, the second 1, the third 2, etc.) are detected and cause no additional overhead.

### 10.1.3  Record Representation Clauses

Record representation clauses are implemented to the bit-level. Records containing discriminants and dynamic arrays may not be organized as expected because of unexpected changes of representation. There are no implementation generated names that can be used in record representation clauses.

Record representation clauses allow more precise packing than pragma **Pack**. Record representation clauses allow the user to specify the exact location of fields within a record to the bit-level. The ICC Ada compiler implements bit-level record representation clauses including nested records starting on bit-boundaries. Since the user specifies the exact bit location, overhead for extracts and stores *may* be very high, so record representation clauses should be applied very carefully. Record representation clauses are implemented using the following rules:

- Fields of records may be allocated to the nearest bit for elements which are smaller than 32-bits. This includes small nested records. Elements which are 32-bits or larger (and all arrays) *must* be placed on byte boundaries.

- If the specified storage space for an element is not adequate using its default allocation, it will automatically be *packed* in two stages: (1) normal packing will be attempted using the default alignment rules. If this does not adequately reduce storage then (2) bit-level packing will be attempted with all fields aligned on 8-bit or smaller boundaries. If this bit-level packing still does not meet the storage requirement, an error message will be generated.

- The optional alignment clause may be used to specify an alignment up to 8192 bytes. However, records can only be aligned to 16 bytes when used within a local stack frame. If a record with an alignment larger than 16 is allocated in the local frame of a subprogram, an error message will be generated.

- All fields of a record representation clause which are left unspecified will be allocated at the *end* of the record using the default alignment rules for each element.

- The fields of a record representation clause may be specified in any order and the storage order of the fields does *not* need to be the same as the order in which they were declared.

- If no alignment clause is specified, the alignment requirement for the record is equivalent to the largest alignment requirement of its elements.

## 10.2   Address Clauses

Address clauses are implemented for variables. Address clauses for local variables using dynamic values are implemented. The use of a dynamic address can facilitate overlaying since the address specified may be the value of a variable of type System.Address or may be the result of an expression using the predefined Address attribute. Address clauses are not implemented for subprograms, packages, tasks, constant objects, or statement labels.

# 11   Interface to Other Languages

Pragma Interface allows Ada programs to interface with (i.e., call) subprograms written in another language (e.g., Assembly); pragma Export allows

programs written in another language to interface with programs written in Ada. The accepted languages are: Intrinsic, Ada, C, and Assembly. The aliases Assembler and ASM can be used instead of Assembly. The alias Builtin can be used instead of Intrinsic. The language Intrinsic should be used with extreme care—it is used by ICC for internally handled operators.

# 12   Unchecked Type Conversion

The generic function Unchecked_conversion is implemented. In general, Unchecked_conversion can be used when the underlying representations of values are similar.

Acceptable conversions are:

- Conversion of scalars. Unchecked_conversion can be used to change the type of scalar values without restriction. In most circumstances the unchecked conversion produces no additional code.

- Conversion of static constrained structures. Constrained static arrays and records are represented as contiguous areas of memory and, hence, can be converted using unchecked_conversion.

- Conversion of scalars to static constrained structures. Scalar objects may be converted to static constrained structures with no additional overhead. If a scalar value is converted to a structure, an aggregate is first built to hold the scalar value and its address is used as the address of the resulting structure.

Because the representation of dynamic structures uses implicit pointers and dope-vectors, ICC Ada does *not* allow unchecked conversions to or from dynamic or unconstrained structures (arrays or records). A compile-time error message will be generated for such instantiations.

Although the Ada compiler does *not* produce errors for the following unchecked conversions, they should be avoided since their results are not obvious:

- Conversion from constrained discriminant records. Conversion from discriminant records can cause unpredictable behavior because of underlying representation changes. Unchecked_conversion will use the

same rules as described above for performing the copy; however, the results of this operation may not be what the user desires since ICC Ada does *not* place arrays constrained by the discriminant in-line with the other fields in a discriminant record. In place of the array only a pointer is used and the array is allocated dynamically from the internally maintained heap.

- Conversion to or from pointers to unconstrained arrays. Unconstrained array pointers are implemented as special dope-vectors in ICC Ada. Conversions to or from these dope-vectors are not recommended.

- Conversion to or from any type or object declared in a generic. Generics can cause hidden representation changes. Unchecked_conversions of any object or type declared in a generic should be avoided.

ICC Ada does not require that the sizes of the parameters to an unchecked_conversion be identical. The size of the *target* type is used to determine the number of bytes to copy. The size of the *target* type (in bytes) is determined by the Ada frontend and exactly that many bytes are copied from the source address to the target address. This can cause problems (e.g., memory faults) when the source object is smaller than the target object. For example, using unchecked_conversion to convert a character into an integer will cause four bytes to be copied starting from the address of the character. The first byte copied will be the value of the character, but the values of the remaining three bytes cannot be predicted since they depend on values of variables or fields immediately after the character in memory. If the source object is larger than the target object then only the bytes that will fit in the target object are copied from the source starting at the address of the first byte of the source.

# 13   Unchecked Storage Deallocation

Unchecked_deallocation is implemented. Unchecked_deallocation of structures containing dynamic elements (such as discriminant records with dynamic arrays) should not be performed since these nested structures are not automatically deallocated.

# 14 Machine Code Insertion

The package MACHINE_CODE described in the LRM is not provided or supported. Instead, machine code insertion is implemented using the ICC-defined package I960_MACHINE_CODE. This package defines a variable for each machine register (G0, G1, R0, FP0, etc.) and a procedure for each instruction (ADDI, ADDO, CALL, LDL, etc.). References to these special variables and calls to these special procedures are trapped by the ICC Native Code Generator and converted into the appropriate machine object (register or instruction). This allows machine code to be inserted in any context, including interspersed within Ada statements.

For example, to perform an FMARK instruction in the middle of a subprogram, the following could be written:

```
with i960_machine_code;
use  i960_machine_code;
procedure force_mark_if_needed(test_condition: boolean;
                                      val: integer) is
begin
   if test_condition then      -- If we're supposed to FMARK...
      mov(val, g0);            -- Move VAL parameter into G0.
      fmark;                   -- Force Mark.
   end if;
end;
```

In addition, the following subprograms are declared in package I960_MACHINE_CODE:

**address_index** This function can be used to explicitly build complex addressing modes with an offset of type System.Address for use in the LOAD and STORE instructions. The Base, Offset, Index-register, and Scaling-factor can all be explicitly provided.

**integer_index** This function can be used to explicitly build complex addressing modes with an offset of type Integer for use in the LOAD and STORE instructions. The Base, Offset, Index-register, and Scaling-factor can all be explicitly provided.

**index** This function is identical to address_index.

**label** This subprogram is overloaded as both a procedure and a function. When called as a procedure it will *emit* a label with the corresponding integer ordinal value into the output code. When called as a function it will *reference* the corresponding label. These subprograms are used primarily for branches and loops.

24